

# About UML transformations based on Graph Rewriting Systems

March 13, 2001

Prof. Dr. Uwe Assmann  
Linköpings Universitet  
Department of Computer Science (IDA)  
Programming Environments Laboratory (PELAB)  
S-58183 Linköping, Sweden  
uweas@ida.liu.se

Alexander Christoph  
Forschungszentrum Informatik  
Softwaretechnik  
D-76131 Karlsruhe, Germany  
christo@fzi.de

## Abstract

In this paper we describe a UML transformation framework based on Graph Rewriting Systems (GRS). Transformations described here, address the Design Implementation problem, i.e. restrictions in the implementation language, that must be noticed during code generation. The transformation framework applies graph rewriting rules to a abstract UML model to build a implementation model, which is adapted to implementation specific restrictions. Beside transformation the framework enables the programmer to modify the implementation model so, that changes are propagated into the abstract model. An optimistic approach keeps both models synchronized.

## 1 Introduction

The design implementation gap describes the common problem, that results from trying to implement an abstract model, obtained from problem analysis and object oriented design. The most common implementation languages and frameworks only support a subset of objectoriented concepts. This results in language dependent designs or the need of experts, which are firm not only in the problem domain, but also in the implementation language. The first approach lacks the abstraction proposed by the UML standard, which is an important prerequisite for software support and extension. The latter requires an additional development step, with implications on development time and costs. Besides this, the 'transformation expert' must guarantee that the original business model is implemented correctly, i.e. that all concepts of the design are implemented. This becomes even more difficult, if concepts like design-patterns or optimizations are to be used. Experience shows, that software changes occur not only in the abstract model, but also in the implementation model. This often results in two different models, which roughly describe each others concepts. In this paper we outline a system, that uses a Graph Rewriting System (GRS) to transform abstract UML models to implementation models. GRS rules have the advantage of being easily described in a graphical notation. This offers the possibility to implement user specific rules, enabling the easy adaption to particular implementation frameworks and languages. Beside the generation of implementation models, the framework keeps both models synchronized, enabling modifications not only in the abstract model but also in the implementation model.

## 2 Graph Rewriting for UML transformations

A Graph Rewriting System (GRS) consists of a set of rules, describing transformations on graphs, which consist of nodes and edges. Nodes and edges can be marked with labels. We can think of a graph as being a UML

model, with the nodes being the classes and the edges being the relations among them. A GRS rule consists of two patterns, which describe the occurrence of a certain subgraph in the host graph and its replacement. The subgraph in the host graph of a particular rule is called *redex*. A rule can contain additional information, which describes the *redex* of the rule more detailed. After the *redex* of a rule is identified by the system, the *redex* is replaced by the second pattern of the rule. Additional embedding instructions can be used to exactly describe how the new graph is integrated into the host graph.

GRS's are well understood and algorithms and frameworks for the evaluation of rules and graph modifications exist. GRS's are naturally applicable to UML models, so the rules can be defined as UML graphs. The UML metamodel allows a fairly simple implementation of a GRS, because of the fixed number of node and edge types.

## 2.1 Transformation

The transformation framework consists of a set of rule sets. A rule set contains a number of rules for a particular transformation target. Examples could be: Java rules, ie. rules to resolve multiple inheritance; EJB rules, ie. rules for generating classes for a certain framework; etc. The transformation unit can apply the rules of a certain set in three different ways.

- **Controlled**  
The user identifies the rule to be executed and its *redex*. The system performs the graph transformation.
- **Programmed**  
The user identifies several rules to be executed one after another. The system tries to find a *redex* for every rule automatically. Additional information can be supplied to use certain strategies for rule evaluation. Strategies could describe how often one rule is executed, etc.
- **Automatically**  
The system determines the rules to be executed automatically. Here, only the set of rules is given by the user. The system can use different strategies to execute rules.

However, it must be noticed that automatic and programmed rewriting systems comprise non-deterministic behaviour. Certain execution strategies can be used to face this problem.

A transformation consists of a set of applied rules together with their *redexes*, regardless of the way they were chosen. The evaluation of a transformation results in a new UML model which is consistent with the original model in respect of the transformation rules.

## 2.2 Rules

Transformation rules describe modifications to the host graph. A rule contains two graphs, called left-hand-side (*lhs*) and right-hand-side (*rhs*). The *lhs* graph describes a graph pattern which must be found in the host graph by the execution unit. After a corresponding subgraph is identified, additional information in the rule can be used to further determine its relevance. These informations relate to values of attributes of nodes and edges. When a subgraph is found to be valid for the current rule, this subgraph is deleted from the host graph. This includes all nodes and edges of the *redex* and all edges between the *restgraph* and the *redex*. In the second stage, nodes and edges of the *rhs* graph are added to the host graph. Additional, particular embedding instructions can be part of the rule, describing how to integrate the new elements into the host graph. Because UML models can be seen as graphs, GRS rules can easily be defined in UML notation. This not only allows the use of well-known concepts, but also improves the readability of transformation steps. Beside this it allows the integration of the transformation framework in existing UML design applications.

### 3 Synchronisation between models

In the software development the problem analysis and software design lead to an abstract software model, which is independent from the implementation. This model describes the structure of the software and is the basis for documentation, modification, development and support. To implement such an abstract model, an additional step is required: mapping the abstract model to a particular programming environment. The resulting implementation model realises the concepts from the abstract model in the programming environment. The experience with large software projects show, that modifications of the software occur not only in the abstract model, but also in the implementation. This leads to two diverging models which are not usable for further developments or support. In most cases this leads to a design bounded to particular implementation environment. To solve this problem, the transformation framework should not only generate the implementation model, it should also keep both models synchronized. This can be achieved using an optimistic approach.

#### 3.1 Optimistic modification of the source model

The transformation framework addresses the design-implementation problem. This means, the source model is independent from the implementation language. The implementation model realises the abstract concepts in the programming environment. Modifications of the implementation can affect implementation specific structures or structures independent from the implementation. Changes to implementation specific structures can be recorded and added to the transformation steps. These structures have no direct representation in the abstract model, therefore no changes to the abstract model are necessary. Modifications of structures that have a representation in the abstract model must be propagated to the abstract model, to represent the current software structure. This problem is solved with an optimistic approach. Structures, that are represented in the abstract model are changed in both models. After this change took part, the whole transformation is carried out again. This new transformation step constructs a new implementation model, which is consistent with the changed abstract model. The framework records all transformation runs and their results to enable the step back, if unwanted results occur.

#### 3.2 Implications of the optimistic approach

The optimistic approach propagates changes made in the implementation model back to the abstract model, if structures are affected, that are defined in the abstract model. After these changes, the whole transformation is executed again. Changes in the abstract model can lead to changes of certain redexes, so that some rules cannot be applied any more. In that case, different strategies can be applied. They include aborting the transformation, creating a new transformation or a partial execution. Therefore a rule dependency graph can be used, which describes, which parts of the transformation can be executed successfully.

### 4 Conclusion

The presented system performs transformations on abstract UML models so, that implementation specific models are generated. Therefore it uses a graph rewriting system, which expresses its transformations as graph modifying rules. Besides the generation of an implementation model, the framework also keeps implementation and abstraction synchronized. Therefore an optimistic approach is used. UML models can easily be expressed as graphs, so the GRS rules can be defined using UML notation. Open questions include partial execution of transformations and rule dependency analysis.

### References

- [1] Uwe Assmann. On edge addition rewrite systems and their relevance to program analysis. In *5th Workshop on Graph Grammars and Their Application to Computer Science*, November 1994.

- [2] Uwe Assmann. Graph rewrite systems for program optimization. In *ACM Transactions on Programming Languages and Systems*, 1995.
- [3] Uwe Assmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. GMD-Forschungszentrum Informationstechnik GmbH, 1996.
- [4] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical use of graph rewriting. Technical report, Department of Computer Science, Queen's University, Canada, 1995.
- [5] G. Rozenberg. *Handbook of Graph Grammars and Computation by Graph Transformation*. World Scientific, 1997.
- [6] Albert Zuendorf. *Eine Entwicklungsumgebung fuer Programmierte Graphersetzungssysteme*. PhD thesis, Technische Hochschule Aachen, July 1995.